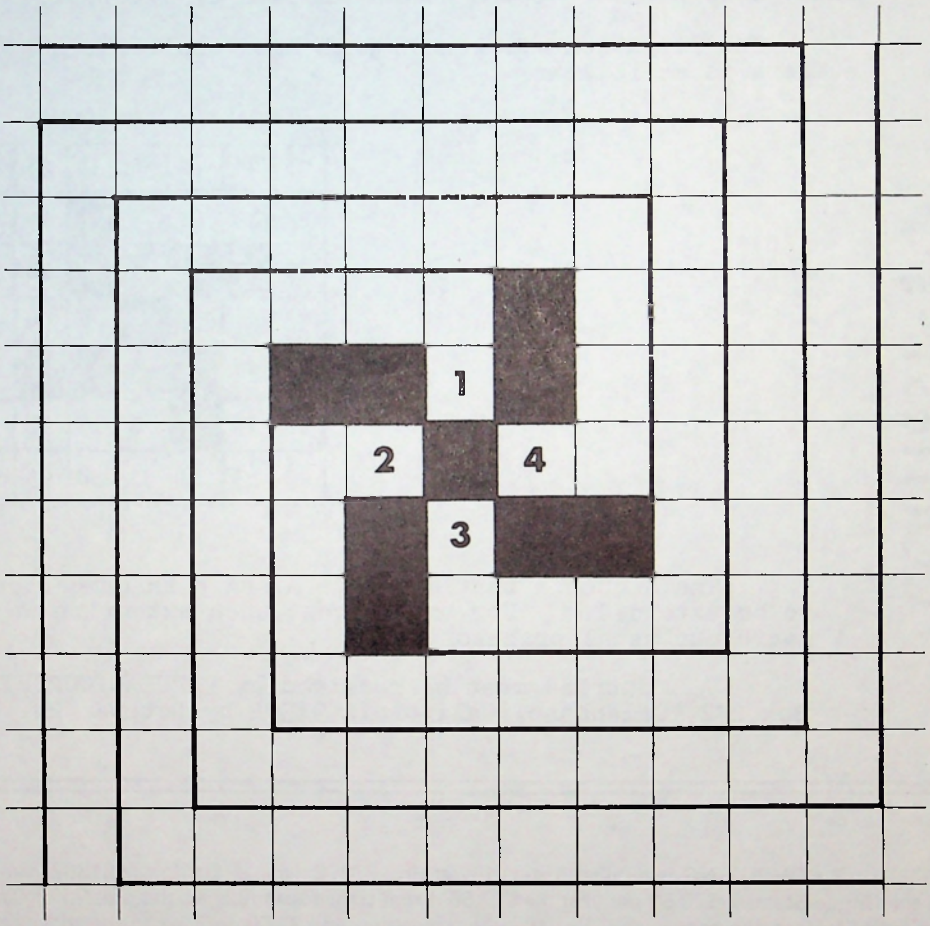
# Popular Computing

41

August 1976 Volume 4 Number 8



## Gammadion

CONTEST 10

PROBLEM 135

In the pattern on the cover there are four paths spiraling out from the center. The numbers that identify the paths are also the contents of the first cell of each path.

The consecutive integers, starting with 5, are to be placed in the paths in rotation. Thus, 5 is placed in the second square of path 1.

If a number has a factor in common with any previously placed number (in a square to its left, right, top, or bottom; that is, orthogonally), it is skipped. Thus, 6 is not the second number in path 2, but 7 is.

The situation after each path has received ten numbers is as follows:

| | | | | | | |
|---|---|---|---|---|---|---|
| 51 | 46 | 41 | 36 | 31 | 26 | 49 |
| 22 | 15 | 11 | 5 | ▨ | 19 | 44 |
| 27 | ▨ | ▨ | 1 | ▨ | 14 | 39 |
| 32 | 7 | 2 | ▨ | 4 | 9 | 34 |
| 37 | 12 | ▨ | 3 | ▨ | ▨ | 29 |
| 42 | 17 | ▨ | 8 | 13 | 18 | 25 |
| 47 | 23 | 28 | 33 | 38 | 43 | 48 |

The sequence that develops along path number one is to be extended. For the longest such extension we will award our usual prize of $25.

All entries must be received by POPULAR COMPUTING, Box 272, Calabasas, California 91302 by October 29, 1976. ☐

POPULAR COMPUTING is published monthly at Box 272, Calabasas, California 91302. Subscription rate in the United States is $20.50 per year, or $17.50 if remittance accompanies the order. For Canada and Mexico, add $1.50 to the above rates. For all other countries, add $3.50 per year to the above rates. Back issues $2.50 each. Copyright 1976 by POPULAR COMPUTING.

Editor: Audrey Gruenberger
Publisher: Fred Gruenberger
Associate Editors: David Babcock
Irwin Greenwald

Contributing editors: Richard Andree
William C. McGee
Thomas R. Parkin

Advertising Manager. Ken W. Sims
Art Director: John G. Scott
Business Manager: Ben Moore

# Dialogue

The following dialogue is taken from letters and notes exchanged between Dr. Richard Hamming and Professor Fred Gruenberger over a period of five years. It began with the following problem in Gruenberger's book *Computing: A Second Course:*

> The astronomy department of a university would like to have the fifth root of 100 (their definition of an "order of magnitude") calculated to 30 decimal places.

. . . with this commentary later in the book:

> The astronomer's problem is marginal. It is a one-shot problem to begin with, so it obviously lacks repetition. Putting it another way, the work of programming a machine for the specified task would serve also for the calculation of *table* of fifth roots. The saving grace is the need for 30 decimal places of precision, which lifts the problem out of the desk calculator class (where it would take perhaps four hours of work) and makes it into a semi-intelligent computer problem.

at which point Hamming wrote in the margin, "False! Guess, divide and average a couple of times to get 10 places; two more multiplications and divisions and you are done."

GRUENBERGER: The theory sounds great, except that I don't recall how to do high precision division on a low precision machine. I can do 30-place multiplications readily by parts, but the method for extending divisors, which I once learned, escapes me. I do recall that it's not easy, which is why the problem quoted from the book is a possible computer problem despite its one-shot nature.

HAMMING: I am ashamed of you! Instead of telling you, I will teach you so you will not forget again. How do you divide in binary by hand? Well, you are apt to tell me that you either subtract the divisor or not, count 1 if you do and 0 if you do not, and then shift one to the right and repeat. Fine. Now, how do you divide in decimal? Well, you could repeatedly subtract, adding 1 each time until you could go no further, and then shift. But that is slow. So instead you guess, by a trial of the leading digits, at the next digit of the dividend, and multiplying the divisor by the trial digit see if you can subtract the product from the current partial number you are dividing. Of course that involved multiprecision multiplication in the sense of a one digit number times a many digit number, and you had to understand "carries." Well, having made the subtraction, you fiddle around to check that your trial digit was neither too large nor too small.

Think out the process definitely, in all its detail. You sure? Now how would you do it in any base, b? Are you sure you have the messy details in mind? If not, they will come to mind as you proceed, so you can recover by going back to the case you do in decimal. Well, if you have a ten decimal digit computer, the base b is $10^{10}$. *You proceed in this base exactly as you did in base 10.* If you have a k binary digit machine (we are in fixed point, of course) then you pick the number base $b = 2^k$ and you proceed. The computer you have can be viewed as having built in the multiplication table in the high number base of single precision. Just fix in your mind that you are in the large number base, and you do arithmetic exactly as you do in the base 10, which is far enough from *two* to be a reliable guide. You, of course, must program the digit by digit arithmetic, include shifts and carries, etc. You can't ask me to write every detail, though I will if you really ask. What is going on is as plain as the nose on your face once you grasp the fundamental point that when you did arithmetic by hand you were doing multiprecision arithmetic over the base 10.

GRUENBERGER: Well, one of us should be ashamed anyway. Now we are agreed on how long division works, and agreed that it is a very simple process – so simple, in fact, that you have not yet done one, like on the 5th root of 100. You were the one who pointed out to me years ago that there are three stages to learning something: study it, teach it; and then teach it to a computer. You have gone through the first two nicely, but I think you're weak on the third. I still want to see you do what you said in the margin of my book was so easy. It isn't. Let me be explicit. We want the root to 30 digits, and you have at your disposal a machine that can do, say, 10 digit arithmetic. Your comment was that after a few divisions, using the Newton scheme, you'd be there. You won't get me off your back until I see you whip this out.

Every day I go hopefully to the mailbox, thinking that today is the day that Hamming will show me how he ran out that 5th root to 30 places but, alas, no letter.

(This would be the proper place to quote from Hamming's article *"Numerical Analysis vs. Mathematics,"* in *Science*, April 23, 1965)

> More generally, when one attempts to put many of the well-known processes of mathematics on a computing machine one finds that there is a great vagueness, and waving of hands, and occasional shouting of "Any fool knows!" and that in the long run a much more careful examination of the basic ideas and processes must be made before one can make much progress. I have been repeatedly shocked to find out how often I thought I knew what I was talking about; but that in the acid test of describing explicitly to a machine what was going on I was revealed to have been both ignorant and extremely superficial. . . . I am also trying to show that the computing expert needs to be wary of believing much that he learns in his mathematics courses; in a sense he must learn mathematics so well that he can defend himself against it.

HAMMING: "Fanatic" is the right word for you on computing. You think that if you can interest the student and get him to be a carbon copy of you that you are doing the right thing. I do not. They should be copies of me, of course! I strongly feel that the entertainment approach to computers produces bad, long term results; the kind of guy you are who thinks that computers are amusing, game playing things. You are proud of the kid who ran two million cases on the A to the B problem – have you no shame that you encouraged a kid to engage in such an anti-intellectual activity? A few cases, yes. But the object is not to get the answer, since it hardly matters; it is to learn to solve problems, of which this is an artificial, but possibly suitable example. You confuse means and ends.

On flowcharts you argue like a child. The fact that flowcharts are needed, or are at least quite useful, *on some problems* is not up for argument – it is your insistance that on *all* problems you should use a flowchart. Your several paragraphs never get to the point of proving this. Indeed, I do not see how you could. We claim that flowcharts have their place among other tools for getting a problem under control, but that they are also not the sole way to truth.

GRUENBERGER: It's true that I foster puzzle-solving with computers; most real-life problems are dull indeed. But it's false to conclude that that's where I stop. Every semester, I get students in my advanced class, who have supposedly had a semester or two of computing and who will admit to being expert programmers. They do some calculation for me, and I ask the simple question "How do you know that these results are correct ?" and I get that blank look – of course the answers must be correct – who would be such an idiot as to question results from a computer? Their education in computing, as far as I'm concerned, starts right there. It seems to me that it is of no concern what problem they work on – the best problem is the one they enjoy doing – but the principles of good computing apply to all problems.

Incidentally, the guy who ran out two million cases on one of my problems was no kid; he was a senior programmer at an expensive government installation; he was spending your money and mine. But if that's the only way the problem could be solved, what choice was there?

---

HAMMING: I have a problem for your class.

Given a number x, greater than zero, to find a rational approximation p/q with q less than or equal to Q.
Start with p = O, q = 1.
If p/q less than x, increase p by one.
If p/q equals x, stop.
If p/q greater than x, increase q by one and test Q.
Save the smallest $|p/q - x|$ .

If you want the k closest p/q values, start the queue with some large numbers (ordered if you wish to think of them.) Start with (O,1) as the (p,q) values.

When a value $|p/q - x|$ is found compare it with the bottom of the queue value of $|x - p/q|$ . Accept or reject. If accept, "bubble up" to its proper place. You end with the k smallest (supposing more than k tries) errors and the corresponding (p,q).

It's simple to program and avoids continued fractions, elaborate sorts, etc.

GRUENBERGER: Your latest note is on a method of finding fractions that approximate any given number. It bears looking into. As you describe it, if one wants the best fraction that approximates Euler's constant to 8 significant digits, say, one would chew up an awful lot of computer time if one followed the algorithm literally and started with (0,1). It seems to me that you're violating your own rules. In any event it's a cute idea, and we'll try it out on some known cases.

HAMMING: You are still promulgating the idea that the way to learn computing is to compute. How much one can learn about computing without doing it depends in part on what you think computing is. You clearly think that it is running a computer, and I clearly think that running the machine is only part of the problem. Your *fanatical* insistance that it is the whole, or almost the whole, and sole goal, that to understand a computer you must run them, flies in the face of simple observation. I have repeatedly seen programmers of many years' experience who in my opinion have no idea of what computing is all about, and who have no understanding of how they fit into the larger society. Thus I conclude that merely running a computer, like merely taking dope, is not sufficient. Some aspects of computing are probably best learned by a hands off attitude – don't get involved and get confused in the sea of minutiae, like so many of my friends have.

Have been thinking a lot about what I think you should publish in place of made up problems whose only point seems to be that it takes a machine to find the answer. There are a couple of problems that I have seen with regard to number theory about the smallest solutions, and I think that backtracking would provide a way of finding them. Thus, an article on backtracking that applies it to a problem from number theory might be something worth doing. You see, I believe in educating as well as amusing.

GRUENBERGER: As I explained in my "goals" paper, you can never tell what problem will grab what person. For example, why in the world did you decide to work on the Searchlight problem? I would have thought that that would be the last one that would appeal to you. There doesn't seem to be any logic to the way people find one problem appealing and another uninteresting, so by my presenting a large number of problems suitable for computer solution, I figure that I'll reach any student sooner or later. And I certainly agree that educating is also a worthy goal.

HAMMING: Some comments on recent issues I received. You should sign "Towards an A Grade" so the reader will know immediately who is speaking – it wasn't God.

Your problem 67 (Chained Primes): reverse the chain, start at low end and see how far up you can go, requiring that 2p + 1 is a prime.

GRUENBERGER: You were concerned about having no by-line on the "A Grade" essay, saying "It wasn't God." Who says? Seriously, that essay accumulated over the years, going through quite a few updatings, so the authorship would be hard to attribute properly.

You're cute. You object to the "jig saw" type of problem, and then proceed to vitiate your argument by working on some of them. If you'll give me the algorithm by which I can tell just which problem will appeal to which group, I'll make up an issue just for you.

HAMMING: I don't like any number theory problem that is base dependent, or dependent on some digits of some numbers.

GRUENBERGER: Your solution to problem 67 on Chained Primes is your usual simplistic self. You said "reverse the chain, start at low end and see how far you can go, requiring that 2p + 1 is a prime." Sure. That's like the testing procedures that my students write: "Run the program with some data and check to see that it works." Your "method" is already over 12 times as inefficient as the one outlined in my article. And it won't get you (in finite amounts of computer time) to first base. No chain of 7 is known, and a lot of CPU time has already been spent trying to find one. Read the last line: if we're looking for chains of 7, we need examine only every 192nd odd number at the top of the chain.

---

HAMMING: Your Error Amplification problem is closely related to integrals worked out by John Wallis (1616-1703). You can find out the general behavior of the numbers you want by using the expansion for the factorial. It is amusing how you invent problems involving the same numbers that come up in real problems. Either it proves that only a few numbers can occur at all, or else you have a natural talent for mathematics that has been perverted into numerology.

GRUENBERGER: I think you may have missed the point of the Error Amplification problem. There are three notions about computing that I try to promulgate:

(1) The results of a long calculation are unpredictable, *particularly* by the writer of the program.

(2) Computers can do things that cannot be done by other means. Not just things that are long and tedious, but which are simply impossible any other way.

(3) Most long calculations cancel out rounding (and other) errors. Some do not: the errors accumulate and lead to disaster.

HAMMING: Change (1) to read: The results of a long calculation are *often* unpredictable, particularly by the *careless* writer of the program. (2) says, in effect, that just because something can be done, it should be done. And as for (3) : Which ones? Can you tell in advance? How?

GRUENBERGER: You continuously distort my meaning. For example, I said that computers can do things that cannot be done by other means. You then changed it to be: Fred just said that just because something can be computed, it should be. How you get from one to the other escapes me. I didn't say that. I can't endorse "useless" computing, nor can you condemn it, until someone decides just what is useful or useless. My point is twofold:

(1) Computers are new and different – not just high speed versions of older gadgets.

(2) Usefulness is in the eye of the user, and any problem that interests a person is a fine problem to work on, to advance his knowledge of computing if nothing else.

I see nothing wrong with the notion of combining good computing with having fun. If you do, then it seems to me that you're saying that computing must be dull and drudgery. Do you find it so? Or try this: Can you point to any computing problem that I've fostered that you would want to publicly label "useless" – and have that opinion hold good indefinitely? Let me remind you of statements in number theory books of the 30's to the effect that arithmetic done in bases other than 10 is interesting but of no practical value.

You say that you want to convert me from useless to useful computations. Take some time to compose working definitions of those two terms. I can't.

HAMMING: I may not be justified, but I do condemn useless computing – by definition, if it is useless! Of course, who can infallibly recognize it?

I deny that usefulness is in the eye of the user. I oppose murder even if the man who does it enjoys it.

I agree that there is nothing wrong in combining good computing with having fun. But fun does *not* make good computing.

There are long calculations whose result is predictable. For example:

```
DØ for I = 1, 10^9
N = 0
N = N + 1
```

You get $N = 0, 1$ (endlessly!) for $10^9$ times.

. . . We are reduced to what is probably useful, and what is probably useless, computing. I doubt that even you would recommend running a program that began with 1 and printed out the result of adding another 1 again and again, for a billion steps. On the other hand, an accurate simulation of the behavior of a proposed design of an atomic bomb seemed to me to be worth all the effort, and I suspect that you also feel that it was; there was no possibility of a small scale experiment. So we agree that there are useful computations.

I claim that you frequently *seem* to say that because a computation can be done and someone wants to do it that it therefore should be done. I deny that people have the right to waste wealth even if they think that it belongs to them – and the courts have on occasion so ruled in my favor. You may not spend your wealth as you please, simply destroying things. Thus, buying up a lot of famous paintings with the intention of burning them will get the courts down on you and prevent your destruction of wealth. Nominally one has control of some wealth whilst he is on earth, but the control is *not* absolute. Well, computing capacity is a wealth, and I deny that the ability to pay for machine time will always justify the expenditure of it even if you want to do it. There are some restraints of reasonable prudence in the disposal of wealth that must be observed.

What I have just said goes double for the wasting of human time. You may not do as you please. Particularly a pied piper like you has not the right to lead children where they love going; you have some obligations to society for their time, attitudes, and the machine time you nominally control. The argument that someone enjoys doing something is not sufficient justification for doing it. There are well recognized rights of others on this earth. Given a finite life and finite energy and finite resources, what is a reasonable way of using them? That is the problem; not what is the right way or the wrong way. What is a reasonable computation and what is an unreasonable one? Foolish, if you wish (after all, my program to print out the integers may well have shown up a failure of the print mechanism that had not been found before). Not that I know infallibly what is good for humans, but often there are probably good things to do – especially with computing machines.

GRUENBERGER: You are a slippery character when you argue! I was talking about usefulness, and said that it lay in the eye of the user. You promptly came up with murder as a counter example. The subject was usefulness, not immorality, or criminality, or whatever Hamming doesn't approve of. Even murder comes under my statement – the murderer can consider it useful, and I'll bet most of them do just that.

But you have distorted the argument in another way. You know perfectly well what I have in mind: the engineering student who balks at the notion of using a computer to count the words that Shakespeare used, or the English student who sees no value in an improved algorithm for calculating logarithms. It's a case of one man's meat and another man's poison. Look back on the last dozen problems that have consumed your time. To you they're useful – would everyone agree? And all of this is still not the main point, which is that I'm out to get people to learn how to compute, and to compute efficiently and effectively (which is two entirely different things). If they work on the finger exercises that most of our computing texts are loaded with, they will conclude that computing is a big bore, and not worth working on. Or, that the speed of the machine can mask all kinds of sloppy thinking, so why bother? It is my contention that through clever problems that *require* the computer for solution, people can be gently led into exploring the niceties of the art – they may even get to consult Hamming's book, or one of min, to find out how others go at it. Just how in hell did we get so smart about computing, anyway? We did a lot of it. I worked on high precision arithmetic, and other useless things; you worked on whatever it is you work on that you find enjoyable, and both of us learned; in my case at least slowly and painfully. I'm trying to shortcut the pain and time lapse for the youngsters. Just what would you have me feed them? Three-dimensional heat transfer problems? New ways to juggle the phone company accounting system to increase revenues? How to penetrate the time-sharing system so as to be able to wipe out everyone's files? Or what?

I would regard as useless *to me* to write a program to generate the numbers from 1 to 100 and add them up one at a time. But that wasn't useless to one of my second course students a few years back. He spent the entire semester on it for a term project. The problem lacked depth (and I downgraded him on that score), but it represented the limits of his computing ability and was useful to him in his learning. (In all fairness, he was not a math or computing major, and he did fairly well at the theory, but he couldn't program a darn.)

I agree that we don't have complete freedom to waste our own material wealth or our time, and there are certainly moral obligations to husband both of them. I *don't* lead children where they love going, as you put it – far from it. If I let my kiddies alone, they'd play Star Trek all day, or write Tic-Tac-Toe programs; the better ones would be off writing endless read routines. I think that computing per se is a useful thing; learning it is difficult for most people, but that chore can be eased; there is no reason why the task cannot be made mildly entertaining and fun. Not everyone enjoys working with differential equations.

HAMMING: The point is that the murderer may regard his action as useful, but I do not accept his opinion. Similarly with respect to useful computing, just because *you* say so does not make *me* accept. A lot of your problems do not appear to be useful to anyone. I would have you pick problems that would appear to be useful to someone; it's not easy to do, but worth the effort. □

# ART OF COMPUTING 12

Until 1960, much work and ingenuity was expended in the writing of symbolic assemblers. In the very early days, these were produced by users; the outstanding example being SAP for the IBM 704 by Roy Nutt. For some machines (e.g., the IBM 650) several assemblers were produced, the most notable being SOAP, which not only did the normal chores of an assembler, but optimized the machine code for the drum storage of the 650.

Later, it became standard operating procedure for the vendor of a machine to furnish an assembler, and that policy has maintained to this day. As a consequence, most existing assemblers were written by a vendor's systems programmers, and the resulting quality level usually leaves something to be desired. With the rise of higher level languages and compilers for those languages, research into assembly systems has attenuated greatly.

A good case has been made for encouraging the use of high level (compiler) languages. Such languages are more powerful, and hence greatly increase programmer productivity in terms of checked out instructions per day. Programs written in high level languages are portable; that is, they can be run on different machines with only minor changes. The languages generally provide improved facilities for debugging and testing and, overall, tend to shorten the elapsed time to production runs of a program. For most systems work and much applications work, there is little question that the use of high level languages is to be encouraged.

But a good case can also be made for the use of assembly languages--they should not be written off just yet. Consider these points:

1. Assemblers are close to the machine they are written for, and permit direct control of machine functions, and this capability is vital to some problem situations. For example, some problem solutions call for information processing at the bit level and some high level languages--Fortran, for instance-- do not lend themselves to such manipulation readily.

2. Carrying this notion further, assembly languages allow the user to capitalize on machine peculiarities. For example, in a problem in which the distinction between odd and even integers is crucial, an assembler permits the isolation of the low order bit of an integer, and a zero test then distinguishes between odd and even. The corresponding manipulations in compiler languages are frequently awkward and inefficient (i.e., wasteful of machine time).

3. For problem solutions that intrinsically demand long machine runs, assembly language <u>can</u> lead to tighter code and hence conservation of CPU time. To be sure, the same end can usually be achieved by proper coding in a high level language; after all, both assembler and compiler language have the same end goal, which is to create machine language instructions. In most situations, the two approaches reach the same goal but by different routes.

4. There is the matter of taste and aesthetics--the feeling "at home" that leads some programmers to prefer working in assembly language. For problem situations that stand alone (that is, that do not interface with other programs), assembly language might be appropriate. Keeping in mind that the end result is always machine language, the programmer who prefers the assembly language route in such situations deserves access to an assembler. It should be noted that an assembler is still available for every machine.

5. In fact, assemblers are returning to prominence with the rise of the micro machines, since these machines are usually so limited in central storage that they cannot sustain compilers.

With all this in mind, let us list the characteristics and capabilities that could be built into an assembler:

1. Mnemonic op-codes, such as LDA for LOAD ACCUMULATOR. Most assemblers make these codes three letters long and, where no ambiguity results, use the first three letters of the meaning of the code. There is no standard practice here, and no forgiveness in any assembler. It might help the "mnemonic" quality of the assembler if it would accept several versions of the same operations, such as

```
DIV
DVD
DVA
D
```

for the operation DIVIDE. The commonest operations might well be single letter or 2-letter codes:

| | |
|---|---|
| L | LOAD ACCUMULATOR |
| A | ADD |
| ST | STORE ACCUMULATOR |
| M | MULTIPLY |
| D | DIVIDE |
| SU | SUBTRACT |
| SA | STORE ADDRESS PORTION |

with all variations <u>also</u> accepted:

| | |
|---|---|
| SUB | SUBTRACT |
| RA | RESET ADD |
| CLA | CLEAR AND ADD |
| MUL | MULTIPLY |
| MU | MULTIPLY |

and so on. In other words, more effort should be put into making the assembler work for the user, instead of the other way around. Clearly, the use of

| | |
|---|---|
| AZJ,LT | ACCUMULATOR ZERO JUMP, LESS THAN |

for the operation everyone else calls

| | |
|---|---|
| JMI | JUMP ON MINUS |

is less than wise, although such things are a matter of taste on the part of the writer of the assembler.

2. Symbols for locations and addresses, with specific (and simple) constraints on the formation of such symbols. A possible rule is this: each symbol must be 8 characters or less, with the leading character an English letter, and with no punctuation symbols allowed within the symbol. The following are then acceptable symbols:

```
X
A
TEMP
TEMP27
REF5
SUBR7
```

and the following would be illegal:

```
3REF
REF 2
X/4
REF3.7
```

3. Simple address arithmetic should be allowed in the address portion of an instruction:

| | |
|---|---|
| *+3 | (current location plus 3 words) |
| REF2-7 | (seven words back from REF2) |
| TEMP3+5*B | (with rules for the nature of this arithmetic, and clearly defined precedence of the arithmetic operations) |

4. Detection of ambiguous and undefined symbols. Since the cross-bookkeeping on symbols is one of the chief tasks of an assembler, this feature is intrinsic to the construction of an assembler. The trick is to let the writer of the program know just what he has done wrong (see item 8 below).

5. Pseudo-op-codes. These are instructions <u>to</u> the assembler, including

| | |
|---|---|
| ØRG | Origin--where this portion of the code is to be assembled. |
| BSS | "Block starting with symbol." |
| ØCT | Octal constant |

6. Binary/decimal conversion. The input and output portions of the assembler should allow the programmer to write in decimal, with all conversions to and from binary done automatically.

7. The assembler should make available on output a symbolic listing and a symbol table. The listing should show everything written by the user, side by side with the absolute machine language conversion. The symbol table should show every symbol used in the program in alphabetic order, the location at which it was defined, and every place in the program that refers to it. Both the listing and the symbol table should be printed at the option of the user.

8. Error messages. Every error that can be detected by the assembler should be explained in English, with clear indication of its source. Ideally, such messages should be printed in the listing on the line following the error, such as:

```
AMBIGUOUS SYMBOL IN ADDRESS PORTION
UNDEFINED SYMBOL
DOUBLY-DEFINED LOCATION SYMBOL
ILLEGAL OP-CODE
ILLEGAL CHARACTERS IN SYMBOL
IMPOSSIBLE TO REACH ABOVE INSTRUCTION
```

9. Line numbers and continuity numbers. Number--ing the lines of the symbolic listing is trivial and obvious. Continuity numbers appear this way:

```
REF1   LDA   CØN     Get constant
  +1   SA    *+2     Store in address of Q
  +2   LDA   ZERØ    Get zero
Q      ST    0000    Store zero
  +1   LDA   *-1     Get critical instruction
  +2   SUB   LIM     Subtract limit
  +3   JPL   ØUT     Jump plus out of loop
  +4   LDA   Q       Get critical instruction
  +5   INA   1       Add one
  +6   SA    Q       Store address back
  +7   UJP   Q-1     Jump to DØ block of loop
ØUT   (Continuation of code)
```

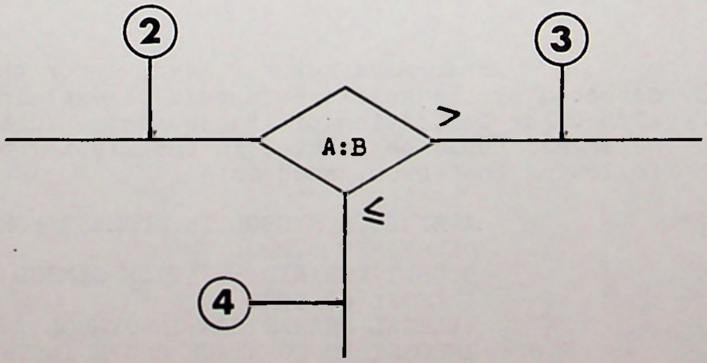
thus making corrections, insertions, and changes easier to make, since the programmer doesn't have to count lines.

10. Comment capability. Liberal provision should be made for the programmer to COMMENT his code.

11. Macro instructions. As system macros, these can be defined as any situation where one line of source code produces more than one line of object code. Consider a machine in which the only conditional jumps are:

| | |
|---|---|
| JMI | Jump on minus |
| JPL | Jump on plus (that is, greater than or equal to zero) |
| JZE | Jump on zero |
| JNZ | Jump on non-zero |

and the following flowchart situation:



after forming the quantity A-B in the accumulator, one yearns for one of the two instructions:

| | |
|---|---|
| JGR | Jump on greater than zero |
| JLE | Jump on less than or equal to zero |

but one can live with (<u>must</u> live with):

```
LØC'N  | ØP    ADDR
-------+-----------
REF2   | LDA   A
       | SUB   B
       | JEQ   REF4
       | JMI   REF4
REF3   |
  .    |
  .    |
  .    |
REF4   |
```

While the <u>machine</u> fails to furnish the desired jump command, the <u>assembler</u> can furnish it, and supply the necessary mechanism. Thus, the assembler supplies more than one instruction per written instruction of the programmer, and the pseudo-instruction JUMP ON LESS THAN OR EQUAL TO ZERO is a macro. A good assembler could furnish many such macros. In the example just given, the operation A:B itself is implemented by forming (A-B) and comparing to zero, in the assumption that the machine lacks a COMPARE command. COMPARE could be another macro.

12. User-defined macros. The concept of macro instructions could be extended to include macros defined by the programmer. Thus, it would be handy to be able to specify a sequence of machine instructions to be called operation XYZ, and from then on in the program the programmer could write XYZ and obtain in the translated program the same sequence each time.

13. Local symbols. Good programming practice calls for segmenting a large program into suitable small parts (modules, if you will) which can each be analyzed, flowcharted, coded, debugged, and tested as independent subroutines. Each such sub-problem will then have a flowchart of its own, and each flowchart may have several reference numbers. It is helpful in following the usual steps in program development to have for these reference numbers small integers, say from 2 to 7. Suppose now that the programmer chooses to use labels (REF2, REF3,...,REF7) that match the flowcharts. The assembler will not normally accept multiple use of the same symbols, unless there is the capability for local symbols. Thus, the programmer should be able to direct the assembler to complete the cross-indexing of symbols up to this point and permit him to re-use any symbols from that point on. (There are, of course, standard techniques to avoid this problem. For example, one can adopt the convention that in subroutine 5, which is entered at label SUB5, all reference symbols be of the form SUB51, SUB52, and so on.)

14. The user of an assembler should be able to specify any of the following options:

a. Assemble and list.
b. Assemble, list, and execute if possible.
c. Execute only--no listing needed.
d. Assemble, list, execute if possible, and cut a binary load deck for future use.
e. Execute and cut a new binary load deck.

(These are properly functions of the operating system, but they appear to the user of the assembler as functions of the assembler.)

Few assemblers have included all of the features listed above, and fewer yet have designed them in satisfactory form; that is, in a form that is human engineered to work <u>for</u> the programmer. The less-than-good assemblers make an unholy mess out of the whole affair, to the point where a programmer (particularly a beginner) spends more time fighting the system than he does on the logic of his problem solution.

Since the construction of assemblers is again a thriving business (for the mini and micro machines), it is suggested that attention be paid to the writing of really good assemblers. □

# CONTEST 5 RESULTS

The problem for our 5th contest (issue number 36, Problem 121) involved taking the natural numbers and removing all those at positions whose numbers are the primes; this removes the primes themselves and leaves a sequence that begins 0, 1, 4, 6, 8, 9, 10, 12, 14, 15,... Then, using that sequence, the same procedure is applied, and this process (remove, from those remaining, the ones at prime numbered positions) is repeated indefinitely. What numbers will survive this sieving process?

The table below shows the first 182 numbers of the desired sequence. This longest list was calculated by Mike Beeler, Cambridge, Massachusetts, who thereby collects the $25 prize.

| | | | | | |
|---|---|---|---|---|---|
| 0 | 6630 | 208328 | 2762589 | 23904247 | 158369991 |
| 1 | 7596 | 228646 | 2977907 | 25499390 | 167754408 |
| 4 | 8676 | 250737 | 3208714 | 27193808 | 177661499 |
| 9 | 9897 | 274753 | 3456018 | 28993079 | 188118314 |
| 16 | 11259 | 300820 | 3720790 | 30903541 | 199152891 |
| 26 | 12784 | 329155 | 4004217 | 32931207 | 210795892 |
| 39 | 14482 | 359909 | 4307547 | 35082690 | 223078698 |
| 56 | 16383 | 393246 | 4632084 | 37365329 | 236034404 |
| 78 | 18502 | 429344 | 4979245 | 39786890 | 249698126 |
| 106 | 20847 | 468446 | 5350401 | 42354798 | 264105759 |
| 141 | 23458 | 510798 | 5747022 | 45077428 | 279295466 |
| 184 | 26354 | 556614 | 6170822 | 47963535 | 295307281 |
| 236 | 29562 | 606184 | 6623497 | 51022255 | 312183335 |
| 299 | 33112 | 659730 | 7106982 | 54263162 | 329967112 |
| 374 | 37041 | 717596 | 7622988 | 57696606 | 348704582 |
| 465 | 41370 | 780071 | 8173639 | 61333058 | 368444185 |
| 570 | 46144 | 847509 | 8761210 | 65183785 | 389236925 |
| 696 | 51401 | 920260 | 9387856 | 69260816 | 411134981 |
| 843 | 57204 | 998667 | 10055938 | 73576698 | 434193748 |
| 1014 | 63575 | 1083172 | 10768151 | 78144201 | 458471679 |
| 1212 | 70566 | 1174233 | 11527105 | 82977246 | 484029536 |
| 1441 | 78251 | 1272312 | 12335740 | 88090079 | 510929941 |
| 1708 | 86675 | 1377836 | 13196950 | 93497884 | 539240533 |
| 2014 | 95920 | 1491389 | 14113972 | 99216792 | 569030899 |
| 2365 | 106029 | 1613472 | 15090138 | 105263802 | 600374208 |
| 2769 | 117082 | 1744694 | 16129003 | 111656271 | 633346440 |
| 3226 | 129173 | 1885740 | 17234332 | 118412705 | 668027769 |
| 3749 | 142389 | 2037270 | 18410171 | 125552452 | |
| 4343 | 156810 | 2199927 | 19660495 | 133096235 | |
| 5016 | 172523 | 2374479 | 20989852 | 141064949 | |
| 5774 | 189664 | 2561736 | 22402772 | 149481827 | |

# Cald's Sequence

In the Journal of Recreational Mathematics, Volume 7, No. 4, page 318, Problem 356, Francis Cald defined a sequence as shown at the top of the facing page.

The column $P_N$ lists successive prime numbers. $F_N$ is $F_{N-1} - P_{N-1}$ if this difference is greater than zero and has not previously appeared. Otherwise $F_N$ is $F_{N-1} + P_{N-1}$ unless that value has previously appeared, in which case $F_N$ is zero. Cald posed these questions:

1. Will every positive integer appear?
2. Will zero appear at all? Or will it appear infinitely many times?
3. What is the rate of growth of $F_N$?

and asked for extended computer runs on the sequence.

A short machine run to calculate the first 240 terms of the sequence showed zero appearing at terms 117 and 199. The accompanying chart shows the scattering of the integers generated. In the short run, the peak value, on the 180th line, was 4601.

So zero does appear. It seems unlikely that every integer will appear. Clearly, longer machine runs should be made on Cald's problem, or an analytic solution should be sought.

| N | $F_N$ | $P_N$ |
|---|---|---|
| 1 | 1 | 2 |
| 2 | 3 | 3 |
| 3 | 6 | 5 |
| 4 | 11 | 7 |
| 5 | 4 | 11 |
| 6 | 15 | 13 |
| 7 | 2 | 17 |
| 8 | 19 | 19 |
| 9 | 38 | 23 |
| 10 | 61 | 29 |
| 11 | 32 | 31 |
| 12 | 63 | 37 |
| 13 | 26 | 41 |
| 14 | 67 | 43 |
| 15 | 24 | 47 |
| 16 | 71 | 53 |
| 17 | 18 | 59 |
| 18 | 77 | 61 |
| 19 | 16 | 67 |
| 20 | 83 | 71 |
| 21 | 12 | 73 |
| 22 | 85 | 79 |
| 23 | 164 | 83 |
| 24 | 81 | 89 |
| 25 | 170 | 97 |

tens and hundreds digits

units digit

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 00 | | X | X | X | X | | X | | | |
| 01 | | X | X | | | X | X | | X | X |
| 02 | | | | | X | | X | | | X |
| 03 | | | X | | X | X | X | | X | |
| 04 | | | | | X | | | | | |
| 05 | | | X | | | | X | | | |
| 06 | | X | X | X | | | | X | X | |
| 07 | X | X | | X | | | X | X | | |
| 08 | X | X | | X | X | X | | | | |
| 09 | | | | | X | | | | X | |
| 10 | X | | | | | | X | | | |
| 11 | | | | | | | X | | | |
| 12 | | | X | | X | | | | | |
| 13 | | | | | | | X | | X | |
| 14 | X | | X | | | | | | X | |
| 15 | | | | | | | | | | |
| 16 | | | X | | X | | X | | X | |
| 17 | X | | | | X | | | | | |
| 18 | | | | | | X | | | | |
| 19 | | | | X | | | | | | X |
| 20 | | | | | | | | X | | X |
| 21 | | | | | | X | | | | |
| 22 | | | | | | | | | | |
| 23 | | | | | | | | | | |
| 24 | | | | | | | | | | |
| 25 | | | | | | | | | | |
| 26 | | X | | | | | | | | |
| 27 | | | | | | X | | X | | |
| 28 | | X | | | | | | | | |
| 29 | | | | | | | | | | |
| 30 | | | | | | | | | | |

The appearance of the integers up to 309 out of the first 240 terms of Cald's sequence. □

# Overdue?--or Done?

Let's list, in no special order, all the brilliant ideas that have advanced the computing art, from the inspiration of the von Neumann machine itself to the present:

- Stored programming
- Closed subroutines
- Delay lines
- Alteration switches
- Addressable stops
- Assemblers
- Compilers
- Interpreters
- Index registers
- Floating arithmetic
- Generators
- Indirect addressing
- Table look-up logic
- Packaged programs
- List processing
- Buffering
- Simulators
- Monitors
- Core storage
- Character-addressable
- Transistorization
- Compiler-compilers
- Parallel processing
- Look-ahead
- Parity checking (redundancy)
- Addressable clocks
- Memory protect hardware
- Trapping
- Tape read integrating circuits
- Read-after-write tape heads
- Byte logic
- Decimal capability
- Convert instruction
- Execute instruction
- Relocatable code
- Reentrant code
- Time-sharing
- Pipelining
- Direct access storage
- Thin film storage
- Microprogramming
- Heuristic programming
- Proprietary programs
- Conversational computing
- Virtual storage
- Structured programming
- Operating systems
- Link editors

You may find your own favorite ideas missing from this list--feel free to add them. It would be difficult to make the list longer than about 60 ideas.

Now count up the man-years that have been spent by people in computing. Figure a handful of pioneers prior to 1950 and perhaps a million or so working full time as computer people today. It would integrate to, say, 2,000,000 man-years.

Doing the indicated division, we reach the conclusion that it takes about 30,000 man-years of effort to bring forth one new idea; this seems to be a rather low yield.

One can immediately argue that other technological industries (automobile manufacturing, for example) have an even lower yield in terms of new ideas per man-year.

But our industry is different. For one thing, it claims to be a brainy industry. Both within and outside the industry, computer people are regarded as possessing above-average intelligence, training, and reasoning power. Artistic we're not, perhaps, but clever, inventive, and innovative we're supposed to be. You might check how many items on that list came before 1965 and how many came after. The last eleven years have not been too fruitful in terms of significant advances.

But a more important point of difference between computing and other disciplines is that in computing a new idea can gain currency more readily. Many new concepts can be simulated on existing equipment to prove their feasibility. The journals are wide open to publication of new ideas; in fact, they are begging for them. Go over the list again; how many of those ideas had to fight their way into existence? The smooth path to acceptance in the computing industry is in sharp contrast to the rocky road for ideas in other technologies. The transistor was invented in 1949, but 27 years later I still hear relays clicking on my telephone line.

What can we conclude from this exercise in gloom? Are we overdue for some new ideas? Or are we done?--there aren't going to be many more new advances? □

| | |
|---|---|
| Log 41 | 1.612783856719735494509411849968180799530513633833687 |
| ln 41 | 3.713572066704307803866763373037407588376410469399302 |
| $\sqrt{41}$ | 6.403124237432848686488217674621813264520420132621019 |
| $\sqrt[3]{41}$ | 3.448217240382730384097423864260789617169992881608157 |
| $\sqrt[10]{41}$ | 1.449700823713563574323657857979012974715908211382102 |
| $\sqrt[100]{41}$ | 1.037833866784767887560987130490313894773441803546342 |
| $e^{41}$ | 639843493530054949.22266340351557081887933662139685527279454959843641678649298486599349 |
| $\pi^{41}$ | 241626620923575111130.28933132294946198857911891573653915908015933859652289097211212 |
| $\tan^{-1}$ 41 | 1.546410917622178082222836679440741930044103777159978 |

Users of the SR-52 and SR-56 programmable calculators now have their own newsletter, 52 Notes. The newsletter is published by the SR-52 Users Club, at a cost of $6 for 6 issues, at 9459 Taylorsville Road, Dayton, Ohio 45424. The first (June) issue summarizes the known pathology of the machine, gives programming tips, and contains two complete programs.

The article "Using New Tools" in issue No. 40 should have been numbered 11 in the Art of Computing series. □